Incremental Integration of Microservices in Cloud Applications

Miguel Zúñiga-Prieto *miguel.zunigap@ucuenca.edu.ec*
University of Cuenca
Cuenca, Ecuador

Emilio Insfran *einsfran@dsic.upv.es*
Universitat Politècnica de València
Valencia, Spain

Silvia Abrahao *sabrahao@dsic.upv.es*
Universitat Politècnica de València
Valencia, Spain

Carlos Cano-Genoves *carcage1@inf.upv.es*
Universitat Politècnica de València
Valencia, Spain

Abstract

Microservices have recently appeared as a new architectural style that is native to the cloud. The high availability and agility of the cloud demands organizations to migrate or design microservices, promoting the building of applications as a suite of small and cohesive services (microservices) that are independently developed, deployed and scaled. Current cloud development approaches do not support the incremental integration needed for microservice platforms, and the agility of getting new functionalities out to customers is consequently affected by the lack of support for the integration design and automation of the development and deployment tasks. This paper presents an approach for the incremental integration of microservices that will allow developers to specify and design microservice integration, and provide mechanisms with which to automatically obtain the implementation code for business logic and interoperation among microservices along with deployment and architectural reconfiguration scripts specific to the cloud environment in which the microservice will be deployed.
Keywords: Microservices, incremental, integration, cloud, cloud architectures

1. Introduction

The need to maintain high customer satisfaction by delivering new or customized products and services signifies that development paradigms are changing to the *Continuous Integration* (CI) and *Continuous Deployment* (CD) of software functionality, in which companies offering internet-based services should be capable of providing customers with software functionality on a daily basis [9]. The microservice architectural style has therefore emerged to facilitate CI/CD by affecting the way in which software development teams are structured, source code is organized and continuously built/packed, and software products are continuously deployed [8]. This architectural style proposes the development of a single application as a suite of small and cohesive sets of microservices built around business capabilities, and independently developed, deployed and scaled, thus allowing them to scale their applications, gain agility and get new functionalities out to customers faster [10], [17].

The flexibility in resource management (e.g. processing, memory, message queues) provided by cloud environments is motivating organizations to consider them as their system deployment environment, in which different Infrastructure as a Service (IaaS) or Platform as a

Service (PaaS) environments are chosen depending on the Service Level Agreement (SLA) or other requirements. Cloud environments are a well suited option as regards deploying microservices [14], [2], since they allow companies to gain agility and reduce complexity not only when deploying and scaling microservices, but also by acquiring resources provisioned according to specific microservice needs. However, applications that will be deployed in cloud environments (*cloud applications*) must be developed using cloud-specific standards, thus preventing developers from creating software that can be deployed on multiple clouds, which is known as *vendor lock-in* [6]. The incremental nature of the microservice-based applications development additionally leads to a situation in which the application's architecture evolves each time a microservice is integrated into it. Building microservices for deployment in cloud environments therefore requires managing architectural changes (architectural reconfiguration) and minimizing application disruptions while the integration takes place.

Current cloud development approaches do not support microservice development/migration and only a few technical reports on this can be found (e.g., [2], [16], [19]). Approaches that support the development of cloud applications are related to this work (e.g., [11], [12], [1]); however, proposals confronting the incremental development and its architectural implications are still lacking. Furthermore, in terms of architectural reconfiguration, as far as we know, there are no proposals that support a systematic reasoning about the architectural impact of the integration of services included in a given software increment into the current application architecture. In previous works [23],[25], we introduced a general process definition for the DIARy method which follows an incremental and MDD approach that supports the incremental integration of cloud service applications and their dynamic architecture reconfiguration triggered by the integration of new *software increments* (hereafter referred to as increments); to support the specification and generation of some software artifacts for service architecture reconfiguration. In this paper, we extend the DIARy process by defining new activities and tasks to satisfy microservices principles and support the incremental integration of microservices. We also provide the tool support needed to automate these tasks by defining the metamodels, which define the microservice integration logic, as well as the transformation chains, which automate the generation of software artifacts that implement the integration logic (orchestration among microservices), and scripts for architectural reconfiguration.

The remainder of this paper is structured as follows. Section 2 contains a description of the background and discusses related works. Section 3 presents an overview of the method proposed. Section 4 illustrates the use of our approach in a case study. Finally, Section 5 presents our conclusions and future work.

2. Background and Related Work

The microservice architectural style is a lightweight subset of the Service Oriented Architecture (SOA), in which: "*the main difference between SOA and microservices is that the latter should be self-sufficient and deployable independently of each other while SOA tends to be implemented as a monolith*" [21]. This architectural style is gaining acceptance as regards overcoming the shortcomings of a monolithic architecture in which, rather than having the application logic within one deployable unit, applications are decomposed into services, each of which is deployable on a different platform, runs its own process, and communicate by means of lightweight mechanisms. The main principles of microservices are [10]:

1. *Componentization via Services*: Software is broken up into multiple services that are independently replaceable and upgradeable and communicate by means of inter-process communication facilities using an explicit component-published-interface.
2. *Organized Around Business Capabilities*: Microservices are implemented around business areas, in which services include a user-interface, storage, and any external collaborations.
3. *Products not Projects*: Development teams own a product throughout its entire lifetime, taking full responsibility for the software in production.

4. *Smart Endpoints and Dumb Pipes*: Business logic, related business rules, and data reside in the services themselves rather than in a centralized middleware. Simple messaging or a lightweight messaging bus is used to provide communication among microservices.
5. *Decentralized Governance*: Standardization on a single technology platform is avoided; the right technological stack for a job should be used, and each microservice manages its own decisions regarding tools, languages, and data storage.
6. *Decentralized Data Management:* Decisions concerning both the conceptual model of the world and data storage will differ between microservices.
7. *Infrastructure Automation*: Automatic means to integrate and deploy in new environments.
8. *Evolutionary Design*: Services are independently replaced and upgraded, which is achieved by using service decomposition as a tool so as to enable application developers to control changes in software applications at the pace of business changes.

Decentralized Governance and *Decentralized Data Management* microservice principles suggest avoiding standardization in a single technology; however, certain development challenges (e.g., the *vendor lock-in*) need to be addressed in order to produce services that are feasible for deployment in different cloud environments. Furthermore, the *Infrastructure Automation* microservice principle suggests having an automatic means of integration and deployment in new environments. However, despite the fact that development teams building microservices use CI/CD techniques and tools[10], these techniques require the inclusion of reliable software artifacts (e.g., implementation code, deployment scripts, configuration scripts) in their automated building processes or deployment pipelines. Software artifacts should therefore be error free in order to ensure that the CI/CD's automated test functionalities do not prevent the integration or deployment process. Finally, CI/CD requires the making of architectural decisions [21], and in a context in which the application architecture evolves with each microservice integration, mechanisms that support the specification of architectural decisions and manage architectural changes without preventing the execution of applications are therefore required.

Model-Driven Development (MDD) is an approach used to develop software systems in which developers build an application by refining models at different levels of abstractions, and then obtain implementation artifacts by means of model transformations. We believe that an MDD approach provides good support as regards managing microservice integration and the consequent architectural evolution of the application. This approach will allows developers to: i) capture technology-independent microservice integration specification and deployment information, thus making design artifacts reusable and enabling developers to overcome the *vendor lock-in* issue; ii) propagate microservice integration specification to implementation/deployment/reconfiguration artifacts, thus enabling developers to obtain error free artifacts; and iii) automate building, packaging, deployment and the architectural reconfiguration process.

Related Work

Developing applications by using the microservice architectural style is a relatively new approach, and only a few related technical reports can be found (e.g., [2], [16], [19]). These works describe design decisions made or strategies employed in order to either satisfy microservice principles, or make use of CI/CD tools and techniques; however, they do not propose design, implementation or integration methods. Moreover, those works do not propose mechanisms with which to help obtain error free artifacts to be included into CI/CD pipelines.

Microservices are cloud-native architectures, and the MDD approaches that support the development of cloud applications are therefore related to this work (e.g., [12], [11], [1], [20]). These approaches apply MDD principles in order to tackle the *vendor lock-in* problem when developing or migrating cloud applications. With regard to approaches that propose mechanisms with which to document design decisions in cloud environments we can highlight CAML [3], MULTICLAPP [13] and CloudML [4]. These works define UML profiles or other modeling languages used to describe deployment topologies, applications as a composition of software artifacts to be deployed across multiple clouds, or resources that a given application

may require from existing clouds. However, although "getting integration right is the single most important aspect of the technology associated with microservices" [17], these proposals do not provide mechanisms with which to specify architectural decisions regarding integration and the impact of integrating increments in the current cloud application architecture. Finally, with regard to approaches for dynamic reconfiguration, works such as SeaClouds [5] or MODAClouds [1] propose mechanisms that can be used to achieve architectural reconfiguration either by replacing orchestration or as result of the re-deployment of components. These proposals do not allow the specification of the architectural changes produced during integration nor do they take into account implementation alternatives that facilitate scalability and the re-deployment of services in different clouds.

3. A Method for the Incremental Integration of Microservices

This method allows cloud applications to be constructed as a composition of microservices, in which each microservice design is included in an incremental increment integration process that allows developers to specify how microservices will be integrated into a cloud application. Developers use the increment integration specification to generate software artifacts, such as skeletons of microservice logic, interaction protocol and scripts with which to build, deploy and architecturally reconfigure the current cloud application, all of which are generated according to each microservice technology specification. In order to define this method, we analyzed how our previous work satisfies the principles of the microservice architectural style; we then used the lessons learned to extend the *DIARy-process* [23],[25]. The *Microservice Incremental Integration Method,* which is made up of the *Microservices Incremental Integration Process* (also referred to as the *Integration Process*), the adapted *DIARy-specification-profile* [24] and transformation chains, is explained as follows. **Fig.1** shows the *Integration Process,* whose main activities are explained in the next sections:

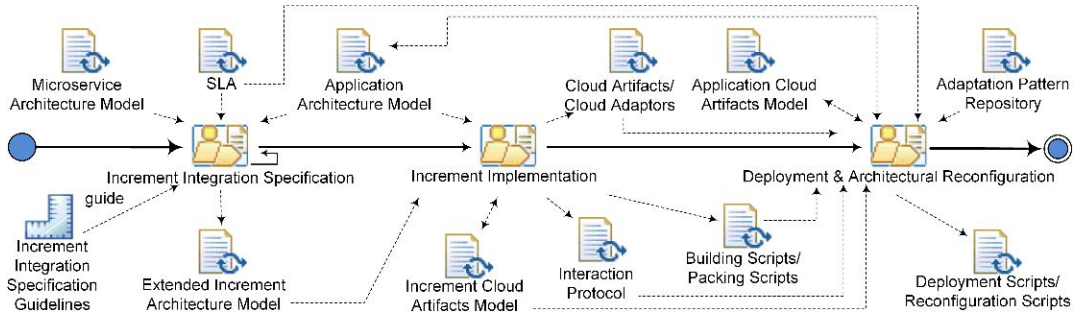

Fig.1. The Microservice Incremental Integration Process

Increment Integration Specification

This activity aims to allow developers to specify how to integrate an increment into the current application (*Application Architecture Model*) by specifying both the integration logic and the architectural impact of integration without taking into consideration the specifics of any cloud environment. In this activity, developers take a microservice (*Microservice Architecture Model*) as input, include it as part of an increment, follow *Increment Integration Specification Guidelines*, and make integration design decisions based on *SLA* terms (whose definition, specification and representation is outside of this work scope). Since this is an iterative activity, it provides developers with the possibility of specifying the integration of increments composed of several microservices. The *DIARy-Specification*-profile (see [24] for more details about its usage) helps developers create the *Extended Increment Architecture Model* which describes the increment specification integration by documenting the increment's architecture, the integration's logic and the architectural impact of integration. This model complies with the *Extended Increment Architecture Model metamodel*, which is explained below.

The Extended Increment Architecture Model metamodel

The *Service oriented architecture Modeling Language* (SoaML) [18] is an OMG specification that was specifically designed for the modeling of service-oriented architectures. SoaML leverages the Model Driven Architecture (MDA) and provides a UML profile and a metamodel that extends the UML metamodel. The *DIARy-specification-profile* extends the SoaML profile, resulting in an ADL that facilitates the increment integration specification. In order to facilitate software artifact generation, this work extends SoaML and UML metamodels in order to define the *Extended Increment Architecture Model* metamodel (see **Fig.2**). Owing to space limitations, **Fig.2** includes only those meta-classes that define the main concepts used to describe integration logic and architectural impact, in which meta-classes belonging to the UML metamodel are depicted with an icon next to the meta-class name, whereas meta-classes that extend the SoaML/UML notations are depicted with a background color.

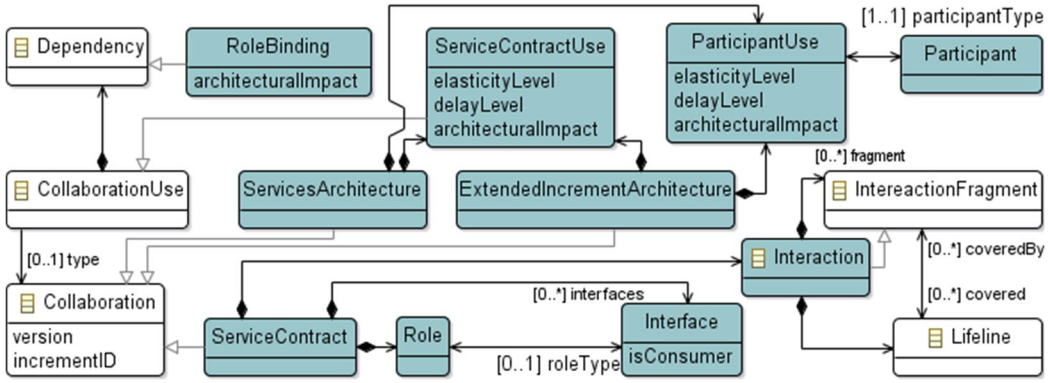

Fig.2. Extract of *Increment Architecture Model* meta-model

A *Participant* represents: i) a microservice to be integrated, ii) a microservice/component already existing in the current *Application Architecture Model* with which the microservice(s) to be integrated will interoperate, and iii) a microservice/component to be created in order to consume microservice services or provide it with services.

The *Services Architecture of the participant* is modeled as SoaML *Services Architecture* diagrams and specifies how parts of a microservice work together to play the owning microservice role(s). This diagram includes the microservice's inner architectural elements and its interoperation requirements.

An *Extended Increment Architecture* extends a UML *Collaboration*, thus allowing the increment integration specification by describing both the integration logic and the architectural impact of integration. Its inner parts (*ParticipantUse*, *RoleBindings*, and *ServiceContractUse*) describe an architecture: the increment's architecture. Each *ParticipantUse* is a reference to a *Participant* involved in the integration. Each *ServiceContractUse* is a reference to a *ServiceContract* that describes interoperation among *Participants*. *RoleBindings,* and *ServiceContractUses* define the integration logic, whereas tagging *ExtendedIncrementArchitecture* inner parts with *architecturalImpact* values defines the architectural impact of integration.

A *ServiceContract* extends a UML *Collaboration*. When modeling the *Services Architecture* of the participant, its semantic remains the same as that defined by SoaML. When modeling the integration (*ExtendedIncrementArchitecture*), it describes specific roles that *Participants* will play. *Roles* have a name and an *Interface* type that specify operations and events, which comprise the interoperation *Interactions* among *Participants* after integration.

A *ParticipantUse* is a *Participant* involved in a specific integration, where the *delayLevel* attribute makes it possible to specifying whether the *Participant* requires immediate processing of requests or whether they could be delayed by using cloud environment services (e.g., message queues). The *elasticityLevel attribute* allows the specification of requirements concerning the level of the *Participant* scaling needs.

The *ServiceContractUse* extends the UML *CollaborationUse* and explicitly specifies the

use of the interoperation described in a Service Contract.

A *RoleBinding* binds each of the *Roles* of a *ServiceContract* to a *Participant*, both of which are referenced in an *ExtendedIncrementArchitecture*.

The attribute *architecturalImpact* allows developers to specify the architectural change (*Add*, *Modify*, *and Delete*) that an architectural element (i.e., *ParticipantUse*, *RoleBinding*, *ServiceContractUse*) will produce on the current *ApplicationArchitectureModel*.

Integration Specification

In order to specify increment integration, developers take the input *Microservice Architecture Model* and create a *Participant* that becomes its owner, and the *Microservice Architecture Model* input then becomes the *Services Architecture* of the *Participant* created. As stated previously, the *Services Architecture of the participant* not only describes the inner architectural elements of the microservice, but also its interoperation requirements, which describe the *outside roles* that external *Participants* must play in order to interact with the microservice along with interaction among those roles by means of the *outside ServiceContracts* also described.

Once the *Participant* representing the microservice to be integrated has been created, developers specify integration logic by creating an *ExtendedIncrementArchitecture* element, and then create its inner parts: i) a *ParticipantUse* that references the already created *Participant*; ii) *ParticipantUses* that reference *Participants* belongings to the current *Application Architecture Model* that will play the *outside roles*, and with which the microservice(s) to be integrated will interoperate; iii) *ServiceContractUses* that specify interoperation among *Participants* by referencing the aforementioned *outsideServiceContracts;* iv) *RoleBindings* that bind each of the roles defined in an *outsideServiceContract* to the *ParticipantUse* that will play the role.

Developers specify the architectural impact of integration by tagging *ExtendedIncrementArchitecture* inner parts with *architecturalImpact* values that describe how they collaborate to reconfigure the *Application Architecture Model* (e.g., by adding *RoleBindings*, adding *Participants*, removing *Participants*). Finally, developers specify information about the management of changes in workload required by each *ParticipantUse* and *ServiceContractUse* by specifying *elasticityLevel* and *delayLevel* values.

By creating *Extended Increment Architecture Models*, which extend UML and SoaML, developers satisfy *Componentization via Services* and *Organized Around Business Capabilities* principles. Furthermore, having *Microservice Architecture Models* as input allow developers to take full responsibility for the software in production, which facilitates incremental integration and satisfies the *Evolutionary Design* and *Products not Projects* principles. Designing microservice integration in advance simultaneously gives different development teams working on different microservices the independence to design, implement and deploy microservices.

Increment Implementation

This activity aims to support the integration process by generating platform-specific cloud artifacts (software artifacts to be deployed on a cloud platform), includes the following steps:

Check Increment Compatibility

Developers participate in verifying whether the *ExtendedIncrementArchitectureModel* is compatible with the current *ApplicationArchitectureModel*. If discrepancies exist between the Participant's interfaces (e.g., different names for methods and services, different message ordering), they design a *ServiceContract* that overrides *outside ServiceContracts* and apply model-to-text (M2T) transformations that generate skeletons of *Cloud Adaptors* (see **Fig.1**).

Specify the Packaging and Deployment Structure

In this step, developers apply model-to-model (M2M) transformations to translate the *Extended Increment Architecture Model* into a model that describes the cloud artifacts needed to implement its inner parts (*DIARyArchitecturalElement*), the *Increment Cloud Artifacts Model*

(see **Fig.1**). This model organizes cloud artifacts into projects that can be packed/built/deployed independently in different cloud environments in accordance with decisions made during the development process (e.g. technology, microservice workload management decisions). This model promotes the decoupling of software artifacts that implement interaction protocol from those that implement microservice design, thus satisfying the *Smart Endpoints and Dumb Pipes* microservice principles. The *Increment Cloud Artifacts Model* complies with the *Cloud Artifacts Model* meta-model (see **Fig.3**).

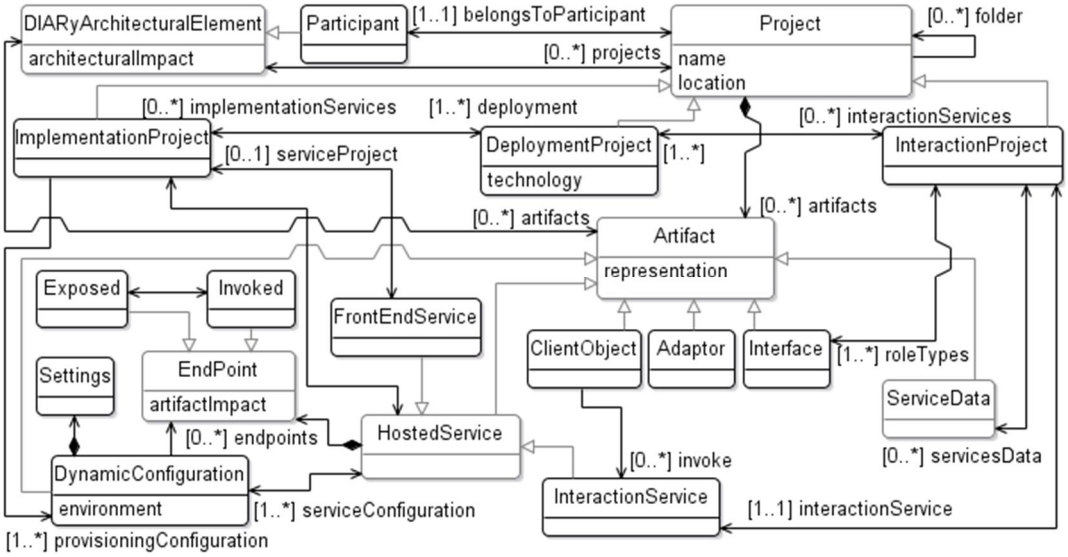

Fig.3. Extract of *Cloud Artifacts Model* meta-model

The Cloud Artifacts Model meta-model

The way in which microservices are deployed has an influence on satisfying *SLA* terms or other nonfunctional requirements [7] (e.g., agility to deploy, modifiability, monitoring, cost of provisioning). We use *Projects* to manage the building, packaging and deployment options. M2M transformation rules map *Interaction Projects* onto *Service Contracts* (see **Fig.2**) architectural elements, and generate descriptions of cloud artifacts that allow developers to implement interoperation among microservices as a separate service. An *Interaction Project* includes the following cloud artifacts: *Interaction Service Hosted Services* that implement interoperation interaction protocols, *Interface* definitions, and *Servicedata* (message Types or data Types). M2M transformation rules map inner parts of *Service Contracts* architectural elements onto these cloud artifacts.

ImplementationProjects, in the case of *Participants* that provide services, are mapped onto *Service Architectures of the participant* architectural elements (see **Fig.2**). These *Implementation Projects* include descriptions of *Artifacts,* such as *FrontEndService Hosted Services* that implement microservice business logic, *Interface* implementations of interfaces defined in related *Service Contracts*, or backend *HostedServices* that use cloud environment services (e.g., message queues). In the case of *Participants* that consume services, *Implementation Projects* are mapped onto related *ServiceContracts* in order to describe *ClientObjectArtifacts* that implement related *Interfaces* and initiate the service execution by invoking *InteractionServices* or *Adaptors* that correct incompatibilities between interfaces. For detailed mappings see [23].

Deployment Projects and *Interaction*/*Implementation Projects* facilitate the packaging of *Artifacts* into a deployable package. Microservices whose related projects are included in a *Deployment Project* will be implemented with the same *technology* and deployed in the same cloud environment, whereas *Artifacts* included in an *Interaction*/*Implementation* project will be packed together in the same deployment artifact and deployed in the same cloud environment resource (e.g., virtual machine), thus sharing cloud environment resources. Including

microservice related *Artifacts* in an exclusive or shared *Interaction/Implementation Project* allows developers to manage workload changes and running costs.

DynamicConfiguration classes describe configuration *Settings* (e.g., *Provisioning Configurations*) that could change at runtime. *Settings* and *Invoked/Exposed EndPoints* information will therefore be stored outside the deployable package. Thus enabling them to be updated without requiring the redeployment of the entire package, a best practice in the CD [15].

In order to *Specify the Packaging and Deployment Structure,* we provide an Eclipse plug-in which executes M2M transformations carried out using the Atlas Transformation Language (ATL) to generate *Increment Cloud Artifacts Models* from *Extended Increment Architecture Models.* Input/Output models are implemented as ecore models in the Eclipse Modeling Framework (EMF). Transformations generate descriptions of the cloud *Artifacts* required to implement architectural elements and define the packaging/deployment structure by assigning *Artifacts* to different *Interaction/Implementation/Deployment Projects* according to the *architecturalImpact*, *delayLevel* and *elasticityLevel* values. **Fig.4** (lines 3, 6, 10) shows an example of the transformation rule applied to assign the *Artifacts* corresponding to *Participants* (e.g, microservices) that require a high *elasticityLevel* into an exclusive *ImplementationProject* (e.g., an exclusive virtual machine) that is assigned to a DeploymentProject (e.g., cloud platform).

```
01.  rule ParticipantUse2Implementation {          -- Create a Implementation project (and related elements) per ParticipantUse
02.  from
03.      ParticipantInput : eiam!ParticipantUse(
04.          ParticipantInput.elasticityLevel = 4 )          -- Filter elements whose elasticityLevel = high (4)
05.  to
06.  implementation : cam!ImplementationProject(          -- Crate an Implementation Project
07.      name <- ParticipantInput.name,                    -- Assign the Participant name to the Project name
08.      belongsToParticipant <- ParticipantInput.participantType,
09.      artifacts <- front,                               -- Create an Artifact FrontEndService
10.      deployment <- thisModule.resolveTemp(             -- Assign Implementation Project to Participant's Deployment project
11.          ParticipantInput.participantType, 'DeploymentProject')),
12.  front : cam!FrontEndService(                          -- Create a FrontEndService element
13.          serviceProject <- implementation),
14.  configimplement : cam!DynamicConfiguration(           -- Create a DynamicConfiguration element
```

Fig.4. Extract of M2M for generating the *Increment Cloud Artifact Model*

Generate Implementation Code

In this step, cloud developers make decisions regarding implementation and deployment technologies that best fit the individual requirements of each microservice included in an increment, and then complete the previously generated *Increment Cloud Artifacts Model* by specifying: i) the *technology* in which each artifact included in a *DeploymentProject* will be implemented and deployed; ii) provisioning, deployment, and inter-service communication information for each *Implementation/InteractionProject,* along with microservice configuration information for each *HostedService*, by creating or updating classes of the *DynamicConfiguration*, *Setting* type (e.g., number of service instances, memory, credentials), or *EndPoint* (e.g., SOAP/REST service style, message format, protocols); iii) the *representation* of each Artifact (e.g., source code language); iv) the *location* of artifacts to be generated.

Next developers execute M2T transformations that use this model and the *Extended Increment Architecture Model* as input in order to generate the cloud artifact implementations, which are organized into a directory structure according to the *Location* specified for each *Project*. The cloud artifacts generated implement (see **Fig.1**): i) an *Interaction Protocol*, ii) software *Cloud Adaptors,* iii) Cloud *Artifacts* that implement microservice logic, APIs that microservices expose, and as many configuration files as *DynamicConfigurationEnvironments* (e.g., development, production), iv) *Building Scripts/Packaging Scripts* to create deployable packages, according to the *DeploymentProject*s*'* structure. Finally, cloud developers complete the cloud artifacts generated and execute packaging/and building scripts to obtain deployable packages.

Deployment & Architectural Reconfiguration

In this activity developers select the adaptation patterns best suited to integrating the increment's architecture into the current application architecture and execute M2T transformations that generate cloud artifacts that operationalize the adaptation patterns according to *Extended Increment Architecture Model* and the *Increment Cloud Artifacts Model*. The cloud artifacts generated are (see **Fig.1**): i) *Deployment Scripts* with which to deploy (and provision) previously generated packages along with the corresponding configuration files, and ii) scripts with which to reconfigure the application architecture, which use architectural impact specification to dynamically update *EndPoints* information stored in the microservice configuration files. Finally, the *Extended Increment Architecture Model* and the *Increment Cloud Artifacts Model* are used as the input for the M2M transformations that update both the current *Application Architecture Model* and the *Application Cloud Artifacts Model* by integrating the corresponding architectural elements and cloud artifact descriptions (see **Fig.1**).

The *Increment Implementation* and *Deploy & Architectural Reconfiguration* activities allow developers to satisfy the *Decentralized governance* and *Infrastructure Automation* microservice principles by providing models that abstract implementation and deployment decisions from technological aspects, and tools that enable developers to obtain software artifacts that can be used as part of CI/CD pipelines.

4. Case Study

In order to illustrate the use of our approach, in this section we present an excerpt of a case study (adapted and extended from [3]). A manufacturing company wishes to improve the technological support given to its dealers, and is considering updating its already existing *manufacturer* microservice by including new functionalities with which to allow dealers to place production orders and obtain the products ordered by means of a shipping service. **Fig. 5a** shows an extract of the current *Application Architecture Model* which will be involved after integrating the Manufacturer's microservice update.

The development team involved in this new requirement used SoaML to model the architectural design of the new *manufacturer* microservice functionalities and produce the *Microservice Architecture Model* (**Fig. 5b**), described as a *Services Architecture*, whose inner parts (e.g., *ServiceContracts*, *Interfaces*, *Roles*) are not shown owing to space restrictions. The *Microservice Architecture Model* includes microservice architectural elements that describe microservice logic and architectural elements that describe microservice interoperation requirements (depicted with a background color in **Fig. 5b**). Note that the *Participants* that are expected to interoperate with the *manufacturer* microservice (other components/microservices that consume *manufacturer* microservice's services or provide it with services) are indicated by the *ParticipantUses* with dashed outlines (i.e., *:Dealer* and *:Shipper*), whereas that internal microservice components are normal *ParticipantUses.*

Fig. 6a shows the *Extended Increment Architecture Model* resulting from the Increment *Integration Specification* activity, in which the *Microservice Architecture Model* (**Fig. 5b)** becomes the *Service Architecture of the participant* Manufacturer, which is referenced by the *manufacturer ParticipanUse* architectural element. The Microservice interoperation requirements described in the *Microservice Architecture Model* (depicted with a background color in **Fig. 5b**), was referenced in the *Extended Increment Architecture Model,* thus becoming the microserivice integration logic (depicted with a background color in **Fig. 6a**).

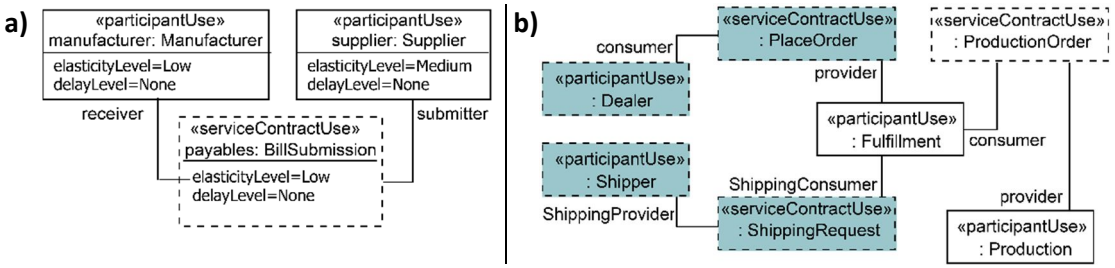

Fig. 5.Extracts of: a) Application Architecture Model, b) Microservice Architecture Model

Developers proceed to specify the *Participants* that will play the roles defined in the integration logic. The *Participant Manufacturer* already exists in the *Current Architecture Model*, and the manufacturer *ParticipantUse* is therefore tagged with *architecturalImpact* = Modify. The *Dealer* and *Shipper Participants* do not exist in the *Current Architecture Mode* and must therefore be created, which is specified by tagging the *ParticipantUses* with *architecturalImpact* = Add. Finally, the requirements for workload change management are specified; in this case study, the new *Manufacturer* functionalities require a High *ElasticityLevel* that differs from the current requirements (see **Fig. 5a**).

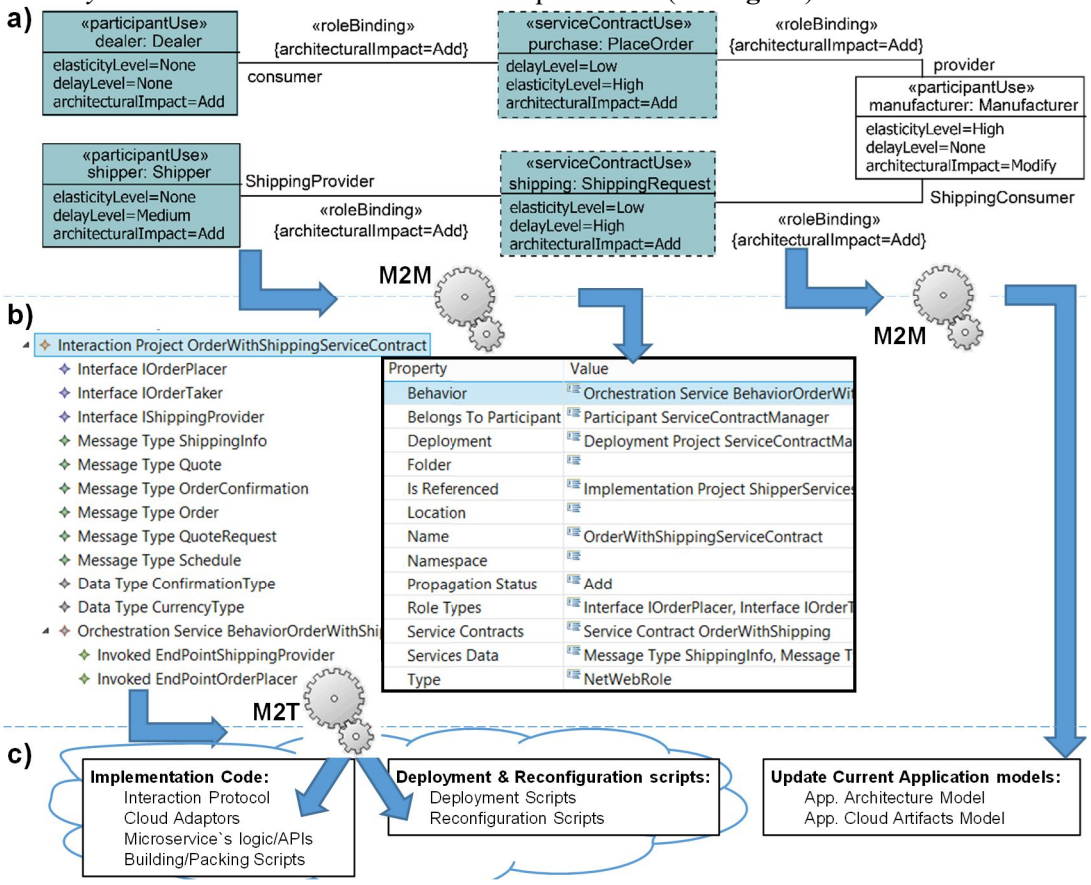

Fig. 6. Main transformation chains a) *Extended Increment Architecture Model*, b) *Increment Cloud Artifacts Model*, c) generated cloud artifacts and updating of current application models

During *the Increment Implementation* activity there were no inconsistencies among *Participants's* interfaces, and the interaction protocols described in the interaction logic were not therefore changed. The *Increment Artifacts Model* (**Fig. 6b**) was generated by applying M2M transformations of the Eclipse plug-in provided with this method, and it was then completed. As the M2T transformations that generate *Implementation Code* (**Fig. 6c**) are not yet finished, we therefore implemented the cloud artifacts required manually, built the application, packed it and deployed it in the Microsoft Azure cloud environment.

During the *Deployment & Architectural Reconfiguration*, we use the open source Eclipse

extension Acceleo M2T generator in order to obtain *Reconfiguration Scripts* (see **Fig. 6c**). We generated XML Document Transform (XDT) files used in Visual Studio to modify service configuration files while the deployment takes place. **Fig.7** (lines 12, 13, 14) shows an example of the transformation rule applied to modify configuration information related to bindings among services in accordance with architectural impact specification.

```
01.  [template public generateElement(aCloudArtifactsModel : CloudArtifactsModel)]
02.  [for(InteractionProjects:InteractionProject | projects->select(oclIsTypeOf(InteractionProject)))]
03.      [file (InteractionProjects.name.concat('/ServiceDefinicion.csdef'), false)]
04.
05.  <?xml version="1.0" encoding="utf-8"?>
06.      <ServiceConfiguration serviceName="[InteractionProjects.name/]" xmlns="…" xmlns:xdt="http://.../XML-Document-Transform" >
07.
08.      [for(IK:Invoked | InteractionProjects.interactionService.endpoints->select(oclIsTypeOf(Invoked)))]
09.          <WebRole name="[InteractionProjects.name/]">
10.              <ConfigurationSettings xdt:Transform="InsertIfMissing">
11.
12.              [comment parseXDT executes a mapping between artifactImpact and XDT values /]
13.                  <Setting name="[IK.name.concat('_EndPoint')/]" [parseXDT(IK.artifactImpact)/]/>
14.                  <Setting name="[IK.name.concat('_Binding')/]" [parseXDT(IK. artifactImpact)/]/>
```

Fig.7. Extract of M2T used to generate *Reconfiguration Scripts*

Finally, the M2M transformations that *update current application models* (see **Fig. 6c**) are in the process of being built; however **Fig. 8** shows how the *Application Model Architecture* is expected to look after integration.

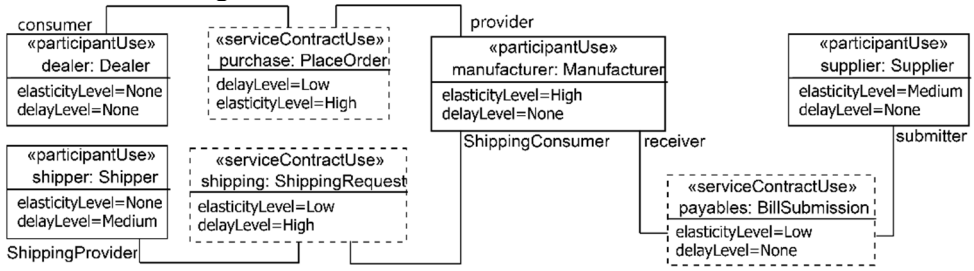

Fig. 8. Current *Application Model Architecture* after integration

5. Conclusions and Future Work

We presented a general view of a method for the incremental integration of microservices into cloud applications. In this method, developers specify how to integrate a microservice into the current application by describing both the integration logic and the architectural impact of integration without taking into consideration the specifics of any cloud environment. They then use both the microservice design and the integration specification to generate: i) skeletons of the microservice implementation code and the integration logic implementation code, ii) scripts to build and package the related microservice software artifacts, iii) scripts to deploy the microservices, and iv) scripts to manage the current application's architectural reconfiguration produced by the integration. Particular emphasis has been placed on explaining how the method manages to keep the microservice design independent from the integration specification, thus allowing different development teams to work on different microservices and giving them the independence to design, implement and deploy microservices according to the implementation/deployment technological requirements of each microservice. Providing developers with tools that automate integration and deployment operations help developers in eliminating discontinuities between development and deployment through CI/CD support which is required in order to deliver new functionalities to customers in an agile manner.

We have shown the feasibility of our proposal by applying it to a case study. We are currently working on implementing transformation chains; however, our approach does not take into account the automation of infrastructure changes. We are considering the use of the DevOps approach in order to improve the collaboration between development and operations, thus allowing new software releases to be made available much faster [22]. In this context, as further work we plan to adapt the method presented in this work in order to satisfy DevOps practices which promote the automation of the process of software delivery and infrastructure

changes. Additionally, even though microservices related artifacts are generated according to architectural impact of microservices' versions, we plan to provide mechanisms to manage incremental consistency, avoiding to lose changes introduced in the implementation code after generation (e.g., changes in interface implementations). Finally, we also plan to design experiments with which to validate the effectiveness of our approach in practice.

Acknowledgements

This research is supported by the Value@Cloud project (MINECO TIN2013-46300-R), DIUC_XIV_2016_038 project, and the Microsoft Azure Research Awards.

References

1. Ardagna, D., Nitto, E. Di, Milano, P., Petcu, D., Sheridan, C., Ballagny, C., Andria, F. D., and Matthews, P., 2012, "MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds," pp. 50–56.
2. Balalaie, A., Heydarnoori, A., and Jamshidi, P., 2015, "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report," pp. 1–15.
3. Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., and Kappel, G., 2014, "UML-based Cloud Application Modeling with Libraries, Profiles, and Templates," in In Proc. Workshop on CloudMDE, pp. 56–65.
4. Brandtzæg, E., Mosser, S., and Mohagheghi, P., 2012, "Towards CloudML, a Model-based Approach to Provision Resources in the Clouds," in 8th European Conference on Modelling Foundations and Applications (ECMFA), pp. 18–27.
5. Brogi, A., Ibrahim, A., Soldani, J., Carrasco, J., Cubo, J., Pimentel, E., and D'Andria, F., 2014, "SeaClouds: A European Project on Seamless Management of Multi-Cloud Applications," ACM SIGSOFT Softw. Eng. Notes, vol. 39, no. 1, pp. 1–4.
6. Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., and Molina, J., 2009, "Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control," Proc. 2009 ACM Work. Cloud Comput. Secur., pp. 85–90.
7. Costa, B., Pires, P. F., Delicato, F. C., and Merson, P., 2014, "Evaluating REST Architectures-Approach, Tooling and Guidelines," J. Syst. Softw., vol. 112, pp. 156–180.
8. Familiar, B., Microservices, IoT, and Azure: : Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions., 2015, Apress.
9. Feitelson, D. G., Frachtenberg, E., and Beck, K. L., 2013, "Development and Deployment at Facebook," IEEE Internet Comput., vol. 4, pp. 8–17.
10. Fowler, M. and Lewis, J., 2014, "Microservices: A Definition of this New Architectural Term." [Online]. Available: http://martinfowler.com/articles/microservices.html. [Accessed: 01-Feb-2016].
11. Frey, S. and Hasselbring, W., 2011, "The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Apps.," Int. J. Adv. Softw., vol. 4, no. 3, pp. 342–353.
12. Guillén, J., Miranda, J., Murillo, J. M., and Canal, C., 2013, "Developing Migratable Multicloud Applications based on MDE and Adaptation Techniques," Proc. Second Nord. Symp. Cloud Comput. Internet Technol. - Nord. '13, pp. 30–37.
13. Guillén, J., Miranda, J., Murillo, J. M., and Canal, C., 2013, "A UML Profile for Modeling Multicloud Applicat.," in Service-Oriented and Cloud Computing, pp. 180–187.
14. Hillah, L. M., Maesano, A., Rosa, F. De, Maesano, L., and Fontanelli, R., 2015, "Service Functional Test Automation," in 10th Workshop System Testing and Validation.
15. Humble, J. and Farley, D.Reliable Software Releases through Build, Test, and Deployment Automation. 2010, Addison-Wesley Professional.

16. Krylovskiy, A., Jahn, M., and Patti, E., 2015, "Designing a Smart City Internet of Things Platform with Microservice Architecture,"3rd Int. Conf. Futur. Internet Things Cloud, pp. 25–30.
17. Newman, S. Building Microservices. 2015, O'Reilly Media, Inc.
18. Object Management Group "Service oriented architecture Modeling Language (SoaML) Specification," 2012.
19. Stefan, B., 2014, "How We Build Microservices at Karma." [Online]. Available: https://blog.yourkarma.com/building-microservices-at-karma.[Access:2-Mar-2016].
20. Vijaya, A. and Neelanarayanan, V., 2015, "Framework for Platform Agnostic Enterprise App. Development Supporting Multiple Clouds," Procedia Comput. Sci., vol. 50.
21. Viktor, F. The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices, 1 edition, 2016, CreateSpace Independent Publishing Platform.
22. Wettinger, J., Andrikopoulos, V., and Leymann, F., 2015, "Enabling DevOps Collaboration and Continuous Delivery Using Diverse App. Environments," pp. 348–358.
23. Zuñiga-Prieto, M., Abrahao, S., and Insfran, E., 2015, "An Incremental and Model Driven Approach for the Dynamic Reconfiguration of Cloud Application Architectures," in 24th Int. Conf. on Information Systems Development ISD2015.
24. Zuñiga-Prieto, M., Abrahao, S., and Insfran, E., 2015, "A UML Profile for Modeling the Integration of Cloud Services in Incremental Software Development (spanish)," in XI Jornadas de Ciencia e Ingeniería de los Servicios (JCIS).
25. Zuñiga-Prieto, M., Gonzalez-Huerta, J., Abrahao, S., and Insfran, E., 2014, "Towards a Model-Driven Dynamic Architecture Reconfiguration Process for Cloud Services Integration," in 8th International Workshop on Models and Evolution (ME 2014), pp. 52–61.